


# Webアプリケーション入門 🚀

フロントエンドとバックエンドの連携

## はじめに：今日やること

- 今日のゴール: 
  - HTTPとは何か
  - GETリクエストとPOSTリクエストの違い
  - サーバーとAPIの役割
  - Fetch APIの使い方

## Webアプリの舞台裏 🎭

- Webサイトやアプリを使うとき、裏では2つの役者が活躍しています。
- それが「クライアント」と「サーバー」です。
- この役割分担を理解することが、開発の第一歩！ 🤝

## 1. クライアント：あなたが見ている「表舞台」 (1/2)

- 役割: ユーザーが直接触れる部分
- 身近な例: Webブラウザ (Chrome, Safariなど)
- 主な仕事:
  - i. サーバーに「お願い (リクエスト)」を送る
  - ii. サーバーからの「答え (レスポンス)」を受け取り、画面に表示する

## 1. クライアント：例えるなら… 🤔 (2/2)

- レストランのお客さん:  
注文（リクエスト）して、料理（レスポンス）を待つ人
- お店のショーウィンドウ:  
商品を見て、店員さん（サーバー）に質問する人
- 関連技術 (イメージ):
  - HTML: ページの骨組み 🦴
  - CSS: ページの見た目 ✨
  - JavaScript: ページの動き ⚡

## 2. サーバー：お願いを処理する「舞台裏」 ⚙️💾 (1/2)

- **役割:** クライアントからの「お願い」に応えるコンピューター
- **場所:** インターネットの向こう側 (データセンターなど)
- **主な仕事:**
  - i. 「お願い (リクエスト)」を待ち受ける
  - ii. データ処理、計算、データベース検索などを行う
  - iii. 結果を「答え (レスポンス)」としてクライアントに返す

## 2. サーバー：例えるなら… 🤔 (2/2)

- レストランのキッチン:  
注文を受けて調理し、料理を出す場所
- お店のバックヤード/倉庫:  
在庫確認や商品準備をする場所
- 関連技術 (イメージ):
  - プログラミング言語 (Python, Ruby, Java...): 処理の手順を書く 📄
  - データベース (MySQL, PostgreSQL...): 情報を保管する棚 🗄️

### 3. クライアントとサーバーの会話 🗣️💬

- インターネットを通じて「HTTP/HTTPS」という共通ルールで会話します。
- 流れ:
  - i. クライアント: 「この情報ください！」 (リクエスト) ➡
  - ii. サーバー: (データを探したり、処理したり...) 🤔
  - iii. サーバー: 「はい、どうぞ！」 (レスポンス) ⬅
  - iv. クライアント: (受け取って画面に表示！) ✨



## 4. なぜ役割が分かれているの？ 🤔 → メリットたくさん！ 👍

- **効率UP:** それぞれ得意な仕事に集中！
  - クライアント → 見た目・操作
  - サーバー → データ処理・保管
- **安全:** 大事なデータはサーバーでしっかり管理 🗝️
- **パワー:** 重い処理はパワフルなサーバーにお任せ 💪 (クライアントPC/スマホは軽く済む)
- **拡張性:** 利用者が増えたらサーバーを増強しやすい 📈

# HTTPを一言でいうと？

インターネットでウェブサイトを見るときの  
「お約束事」や「言葉」のことだよ！

（HyperText Transfer Protocol の略だけど、今は覚えなくてOK！）

# 登場人物をおさらい

登場するのはこの2人！

- あなた (Webブラウザ)
  - Chrome, Safari, Edge など
  - ウェブサイトを見るソフト
- お店 (Webサーバー)
  - ウェブサイトのデータ（文字、画像など）が置いてあるコンピューター

# HTTPの役割：お店での注文に例えると…

ブラウザとサーバーの「会話」をお手伝いするのがHTTPの役割！

## ステップ1：注文する！（リクエスト）

あなた (ブラウザ)

「このページ見たい！」 (リンクをクリック！)

↓

HTTPという言葉で

Webサーバーに お願い (リクエスト) を送る

ブラウザ →→ (リクエスト) →→ サーバー

## ステップ2：料理が出てくる！（レスポンス）

お店（サーバー）

「あいよっ！これが欲しいページね！」（データを用意）



HTTPという言葉で

ブラウザに **お返事（レスポンス）** とデータを送る

ブラウザ ←← （レスポンス：HTML, 画像など） ←← サーバー

## ステップ3：美味しくいただく！（表示）

あなた (ブラウザ)

サーバーから届いたデータを組み立てて…



画面にウェブページを表示！ 🎉

# ちょっと補足：HTTPSって？

最近よく見る `https://` は？

- HTTPを **もっと安全** にしたもの！ (S = Secure)
- 会話の内容（リクエストやレスポンス）を **暗号化** ✨
- 途中で盗み見られたり、書き換えられたりするのを防ぐ！
- 鍵マーク 🔒 が目印！

## Webサイトの裏側の「会話」

- WebブラウザとWebサーバーは **HTTP** という言葉で会話しています。
- **GET** と **POST** は、その会話でよく使う「頼み方」の種類です。



## GET: 情報を「ちょうだい！」とお願いする方法

- サーバーに「この情報を見せて！」とお願いする時に使う。
- **たとえばなら:** 図書館で「〇〇の本はどこ？」と**声に出して**聞く感じ。

# GETの特徴

- お願いごとがURLにくっつく:  
.../search?book=〇〇 みたいにアドレスバーで見える。
- 隠し事はできない: 丸見えなので秘密の情報は送れない。
- 送れる情報量に限りあり: URLの長さに制限がある。
- 何度お願いしても結果は同じ (べき等): 情報をもらうだけ。

# GETはいつ使う？

- 普通のWebページを開くとき
- Googleなどで検索するとき
- 商品一覧や記事一覧を見るとき

➡ 基本的に「情報を見る・もらう」とき

## POST: 情報を「これ、お願いします！」と渡す方法

- サーバーに「この情報を受け取って！」とデータを送る時に使う。
- **たとえるなら: 封筒に入れた手紙（申込書など）を渡す感じ。**

# POSTの特徴

- **送るデータは隠される:**  
URLではなく、リクエストの「中身（ボディ）」で送る。
- **秘密の情報も送りやすい:**  
パスワードなどをURLに出さずに送れる (※別途HTTPSが必要)。
- **たくさんの情報を送れる:**  
長い文章やファイルもOK。
- **お願いすると状況が変わることも:**  
データの新規登録や更新に使われる。

# POSTはいつ使う？

- ログインするとき (ID/パスワード送信)
- 会員登録するとき (個人情報送信)
- ブログやSNSに投稿するとき
- お問い合わせフォームを送るとき
- ファイルをアップロードするとき

➡ サーバーに情報を「送る・渡す・変更を頼む」とき

# GET vs POST まとめ


特徴	GET	POST
目的	情報の <b>取得</b> (ちょうだい！)	情報の <b>送信</b> (お願いします！)
データ場所	URLにくっつく (見える)	リクエストの中身 (隠れる)
データ量	少ない	多い
秘密の情報	不向き	向いている (※要HTTPS)
ブックマーク	できる	できない
繰り返し	結果は同じ (基本)	結果が変わる可能性あり

# どう使い分ける？ (かんたんルール)

- 情報が欲しいだけ？
  - ➡ GET (ページ表示、検索など)
- 情報を送りたい？ 登録・変更したい？
  - ➡ POST (ログイン、フォーム送信、投稿など)
- パスワードとか秘密の情報を送る？
  - ➡ POST (+ HTTPS!)
- 送る情報が多い？ (ファイルとか)
  - ➡ POST



## 大事な注意点：HTTPSについて

- POSTでデータがURLに見えなくても、通信が**暗号化**されていないと途中で盗み見される可能性があります！
- ログイン情報や個人情報などを安全に送るには、**POSTを使うことに加えて、HTTPS** という暗号化通信が **必須** です。
- (ブラウザのアドレスバーに鍵マーク  が出るやつです)

## サーバー と API ってなに？

- **サーバー:**
  - Webサイトの情報や機能が置いてあるコンピュータ。
  - あなたのリクエストに応じてくれる相手。
- **API (エンドポイント):**
  - サーバーが用意している「**リクエスト受付窓口**」。
  - 特定のURL（住所）にアクセスすることで、サーバーの機能を利用できる。
  - 例: `https://api.example.com/users` (ユーザー情報の窓口)

## 通信の道具：Fetch API

- ブラウザに標準で備わっている、サーバーと通信するための機能。
- `fetch()` という関数を使う。
- **非同期処理:** サーバーとの通信は時間がかかるので、JavaScriptは結果を待たずに次の処理に進む。結果の受け取り方が少し特別。

## GETリクエスト：情報を「もらう」

指定したURLから情報を取得します。

**基本的な使い方:**

```
fetch( '取得したい情報のURL' )  
  .then( 最初の返事を受け取る処理 )  
  .then( 実際のデータを受け取る処理 )  
  .catch( エラーが起きたときの処理 );
```

## GETリクエスト：コード例 ( `.then` )

```
const url = 'https://api.example.com/users/1';

fetch(url)
  .then(response => {
    if (!response.ok) { // 成功したかチェック
      throw new Error(`サーバーエラー: ${response.status}`);
    }
    return response.json(); // JSONデータを取り出す
  })
  .then(data => {
    console.log('取得成功:', data); // データ表示！
    // 例: 画面に表示 -> document.body.innerText = data.name;
  })
  .catch(error => {
    console.error('取得失敗:', error); // エラー処理
  });
```

## GETリクエスト：別な書き方 (`async/await`)

`async/await` を使うと、非同期処理が普通の処理のように書けて読みやすい！

- `async function` の中で使う。
- `await` で `Workspace` や `response.json()` の完了を待つ。
- エラー処理は `try...catch` で行う。

```
async function fetchData() {  
  try {  
    const response = await fetch(url); // 待つ  
    const data = await response.json(); // 待つ  
    console.log(data);  
  } catch (error) {  
    console.error(error);  
  }  
}
```

## GETリクエスト：コード例 ( `async/await` )

```
async function fetchUserData(url) {  
  try {  
    const response = await fetch(url); // fetchが終わるのを待つ  
    if (!response.ok) {  
      throw new Error(`サーバーエラー: ${response.status}`);  
    }  
    const data = await response.json(); // JSON変換が終わるのを待つ  
    console.log('取得成功 (async):', data);  
    // 例: 画面に表示 -> document.body.innerText = data.name;  
  } catch (error) {  
    console.error('取得失敗 (async):', error);  
  }  
}  
  
fetchUserData('https://api.example.com/users/1');
```

## POSTリクエスト：情報を「送る」

フォーム入力内容などをサーバーに送信します。

基本的な使い方:

```
fetch('送信先のURL', { // 第2引数に設定オブジェクト
  method: 'POST',      // ① POSTメソッドを指定
  headers: {           // ② ヘッダー情報
    'Content-Type': 'application/json' // 送るデータはJSON形式
  },
  body: JSON.stringify(送るデータ) // ③ 送るデータ本体（文字列にする）
})
.then(返事を受け取る処理) // GETと同じ
.catch(エラー処理);       // GETと同じ
```



## POSTリクエスト：設定オブジェクトの中身

- `method: 'POST'` : これでPOSTリクエストになる。
- `headers` : リクエストの追加情報。
  - `'Content-Type': 'application/json'` : 「送るデータ(body)はJSON形式ですよ」とサーバーに伝える **重要** な情報。
  - 他にも認証情報などを追加することがある。
- `body` : 送信するデータ本体。
  - JavaScriptオブジェクトはそのまま送れないので、`JSON.stringify()` で **JSON文字列** に変換する。

## POSTリクエスト：返事の受け取り方

サーバーにデータを送った後、サーバーからも返事が来ます。

（例：「登録成功しました！新しいIDは123です」など）

受け取り方は **GETリクエストと全く同じ**！

`response.ok` で成功確認し、必要なら `response.json()` などの中身を取り出します。

`.then()` や `async/await` の使い方も同じです。

## POSTリクエスト：コード例 ( `.then` )

```
const url = 'https://api.example.com/users';
const newUser = { name: '鈴木 一郎', job: 'エンジニア' };

fetch(url, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(newUser) // オブジェクトをJSON文字列に
})
.then(response => {
  if (!response.ok) { throw new Error(`サーバーエラー: ${response.status}`); }
  return response.json(); // サーバーからの返事 (例: 作成されたユーザー情報)
})
.then(data => {
  console.log('送信成功:', data);
})
.catch(error => {
  console.error('送信失敗:', error);
});
```

## POSTリクエスト：コード例 ( `async/await` )

```
async function postUserData(url, dataToSend) {
  try {
    const response = await fetch(url, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(dataToSend)
    });
    if (!response.ok) { throw new Error(`サーバーエラー: ${response.status}`); }
    const responseData = await response.json(); // 返事を待つ
    console.log('送信成功 (async):', responseData);
  } catch (error) {
    console.error('送信失敗 (async):', error);
  }
}

const newUser = { name: '鈴木 一郎', job: 'エンジニア' };
postUserData('https://api.example.com/users', newUser);
```

## 注意点：CORSエラー

Workspace を使っていて、コンソールにこんなエラーが出たら？

```
Access to fetch at '...' from origin '...' has been blocked by CORS
```

policy:

- **CORS (Cross-Origin Resource Sharing) エラー**
- **原因:** ブラウザのセキュリティ機能。基本的に、違うドメイン(Webサイト)へのリクエストは制限される。
- **対策:** あなたのJavaScriptコードではなく、**リクエスト先のサーバー側**で「あなたのサイトからのアクセスを許可する」設定が必要。

→ 自分のJavaScriptのコードが悪いわけではない場合が多い！

## 便利ツール：ブラウザの開発者ツール

- ブラウザで **F12** キーを押すと開ける。
- 「**ネットワーク (Network)**」タブを見ると、送受信したリクエストとレスポンスの詳細（ヘッダー、ボディ、ステータスコードなど）を確認できる。
- デバッグ（問題解決）に非常に役立つ！

## まとめ

- Webアプリは「クライアント」と「サーバー」で成り立っている。
- クライアントは「お願い (リクエスト)」を送り、サーバーは「答え (レスポンス)」を返す。
- HTTPはその「会話」のルール。
- GETリクエストは情報を「もらう」ため、POSTリクエストは情報を「送る」ために使う。
- Fetch APIを使うと、JavaScriptからサーバーにリクエストを送ったり、レスポンスを受け取ったりできる。